Lecture 13 - Memory Ordering and Other Atomic Operations

ECE 459: Programming for Performance

Jon Eyolfson

University of Waterloo

February 1, 2012

Memory Ordering

Memory-consistency can also refer to the order of memory operations

- Sequential consistency
  - No reordering of loads/stores
- Relaxed consistency (only some types of reorderings)
  - Loads can be reordered after loads/stores
  - Stores can be reordered after loads/stores
- Weak consistency
  - Any reordering is possible

Reorderings are done if they look safe in the current context (are independent)

Final Exam Question

```
                        x = y = 0

/* thread 1 */                          /* thread 2 */
x = 1;                                  y = x;
r1 = y;                                 r2 = x;
```

Assume the architecture is not sequentially consistent (weak consistency)

Show me all possible (intermediate and final) memory values and how they arise

Final Exam Solution

- You have to go over every permutation of lines (since they can be in any order)
- Then just over all the values
- Won't be on this years final, but shows how memory-reordering could complicate things

Compiler Memory Reordering

The **compiler** may reorder instructions, along with the hardware

Example: we want thread 1 to print a value after thread 2 is done

```
                    f = 0

/* thread 1 */                  /* thread 2 */
while (f == 0);                 x = 42;
printf("%d", x);                f = 1;
```

- If thread 2 reorders its instructions, will we get our intended result?

Compiler Memory Reordering

The **compiler** may reorder instructions, along with the hardware

Example: we want thread 1 to print a value after thread 2 is done

```
                    f = 0

/* thread 1 */                  /* thread 2 */
while (f == 0);                 x = 42;
printf("%d", x);                f = 1;
```

- If thread 2 reorders its instructions, will we get our intended result?
- No

Preventing Compiler Memory Reordering

- A **memory fence** prevents memory operations from crossing the fence
- Also known as a **memory barrier**

```
                        f = 0

/* thread 1 */                          /* thread 2 */
while (f == 0);                         x = 42;
// memory fence                         // memory fence
printf("%d", x);                        f = 1;
```

- This now prevents any reordering

Preventing Compiler Memory Reordering in Programs

- Syntax depends on the compiler

Microsoft Visual Compiler

```
_ReadWriteBarrier()
```

Intel Compiler

```
    __memory_barrier()
```

GNU Compiler

```
__asm__ __volatile__ ("" ::: "memory");
```

Aside: gcc Inline Assembly

Just as an aside, here's gcc's inline assembly format

```
__asm__ ( assembler template
        : output operands                    /* optional */
        : input operands                     /* optional */
        : list of clobbered registers        /* optional */
        );
```

Last slide used **volatile** as well, however this isn't the same as what we've seen before, in this context it means:

- The compiler may not reorder this assembly code and put it somewhere else in the program

Hardware Memory Reordering

An `AMD64` can reorder stores after loads

- Actual details are beyond the scope of this course

`AMD64` class CPUs also have **memory fences**

Preventing Hardware Memory Reordering

Note: these are all `asm` instructions

`mfence`

- All loads and stores before the fence finish before anymore execute

`sfence`

- All stores before the fence finish before anymore execute

`lfence`

- All loads before the fence finish before anymore execute

Preventing Hardware Memory Reordering (Option 2)

- Some compilers also support preventing hardware reordering

Microsoft Visual Compiler

```
MemoryBarrier();
```

Sun Studio (Oracle) Compiler

```
__machine_r_barrier();
__machine_w_barrier();
__machine_rw_barrier();
```

GNU Compiler

```
__sync_synchronize();
```

Relevance to OpenMP

- Fortunately an OpenMP **flush** also preserves the order of variable accesses
- Stops reordering from both the compiler and hardware
- For GNU, it's actually just implemented as `__sync_synchronize();`

Note: the proper use of memory fences makes `volatile` not very useful (again, `volatile` is not meant to help with threading, and will have a different behaviour for threading over different compilers/hardware)

Atomic Operations

- We saw the **atomic** directive in OpenMP
- Most map to hardware instructions that are atomic
- There are other atomic instructions as well...

Compare and Swap

Also called **compare and exchange** (`cmpxchg` instruction)

```
int compare_and_swap (int* reg, int oldval, int newval)
{
  int old_reg_val = *reg;
  if (old_reg_val == oldval)
     *reg = newval;
  return old_reg_val;
}
```

- After, you can check if it returned `oldval`
- If it did, you know you changed it

Implementing a Spinlock

This is essentially the spinlock implementation:

```
void spinlock_init(int* l) { *l = 0; }

void spinlock_lock(int* l) {
    while(compare_and_swap(l, 0, 1) != 0) {}
    __asm__ ("mfence");
}

void spinlock_unlock(int* int) {
    __asm__ ("mfence");
    *l = 0;
}
```

You'll see **cmpxchg** quite frequently in the Linux kernel code

ABA Problem

- Sometimes you'll use read a location twice and assume if the value is the same, nothing has changed
- This is not always true and is an **ABA problem**
- You can combat this by "tagging", there is a double compare and swap which also uses an identifer when swapping

- Just something to be aware of, it won't be tested

Prefix and Postfix

Lots of people use postfix, when really, it should be prefix

In C, this isn't a problem, in some languages (like C++), it can be

Overloading

In C++, you can overload the ++ and – operators

```
class X {
public:
  X& operator++();
   const X operator++(int);
...
};

X x;
++x; // x.operator++();
x++; // x.operator++(0);
```

Common Implementation

Prefix is also known as **increment and fetch**
Postfix also known as **fetch and increment**

```
X& X::operator++()
{
  *this += 1;
  return *this;
}

const X X::operator++(int)
{
  const X old = *this;
  ++(*this);
  return old;
}
```

Efficiency

If you're the least concerned about efficiency you should always use **prefix** increments/decrements instead of defaulting to postfix

Only use `postfix` when you really mean it, to be on the safe side

Summary

- Memory ordering
  - Sequential consistency
  - Relaxed consistency
  - Weak consistency
- How to prevent memory reordering with fences
- Other atomic operations
- A good practice